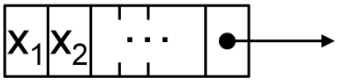
6. Polygonfüllen

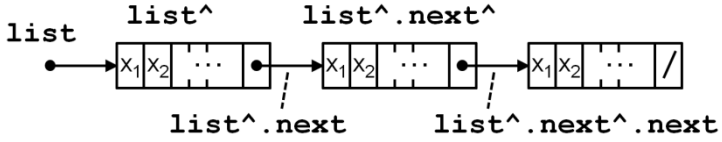
Zum Füllen von Polygonen gibt es drei Klassen von Verfahren. Bei den Scanline-Verfahren wird jede Scanlinie mit dem Polygon (oder mit der Flächenumrandung) geschnitten, und innere Teile (*Spans*) werden gefüllt. Bei den Floodfill-Verfahren wird die Fläche ausgehend von einem Anfangspunkt in alle Richtungen gefüllt, bis eine Grenzbedingung erfüllt ist (z.B. man stößt an eine Grenzlinie). Mit paralleler Hardware kann man für jedes Pixel getrennt berechnen, ob es innerhalb oder außerhalb eines Polygons liegt.

Einschub: Verkettete Listen

Zur Auffrischung wird der Begriff der *verketteten Liste* kurz wiederholt. Eine (verkettete) Liste ist eine Datenstruktur, die es sehr flexibel ermöglicht, Datenknoten in einer bestimmten Reihenfolge zu speichern, und diese Konten einfach umzuordnen, zwischen verschiedenen Listen auszutauschen, zu löschen oder neue Knoten einzufügen. Dazu erhält jeder Datensatz eine zusätzliche Information „Zeiger“ (pointer), der die Adresse des nächsten Knotens enthält. Dies stellt man oft durch Kästchen für die Datensätze und Pfeile für die Zeiger graphisch dar:

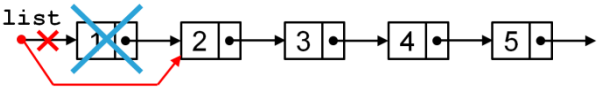


Wir wollen die Zeigerkomponente eines Knotens `next` nennen. Ein Anfangszeiger zeigt auf den ersten Knoten so einer Liste (diesen nennen wir `list`), und ein leerer Zeiger (nil-pointer) bedeutet, dass kein Nachfolger existiert, gibt also das Ende solch einer verketteten Liste an:



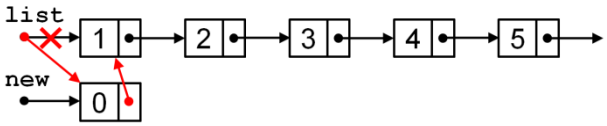
Man kann sich das so vorstellen, dass ein Zeiger die Speicheradresse des Nachfolgeknotens enthält. Diese kann beliebig auch anderen Zeigervariablen zugewiesen werden. Für das Reservieren von neuem Speicherplatz (also dem Erstellen eines neuen Knotens) muss es entsprechende Funktionen geben, die eben auch die Adresse dieses neuen Knotens zurückliefern. Typische einfache Operationen sind (wir bezeichnen mit `x^` das Objekt, auf das der Zeiger `x` zeigt):

Entfernen des ersten Knotens: `list = list^.next`



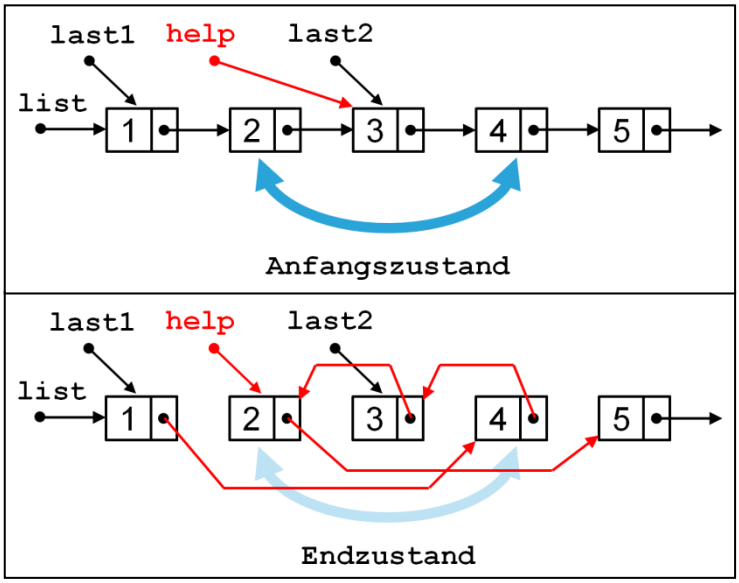
Einfügen eines neuen Knotens am Anfang der Liste:

```
new^.next = list; list = new
```



Vertauschen zweier Knoten:

```
help = last1^.next^.next
last1^.next^.next = last2^.next^.next
last2^.next^.next = help
help = last1^.next
last1^.next = last2^.next
last2^.next = help
```



Für viele Anwendungen sind auch doppelt verkettete Listen praktisch: dabei enthält jeder Knoten nicht nur einen Zeiger auf den nächsten Knoten, sondern auch einen auf den vorigen. Verkettete Listen sind vor allem dann von Vorteil, wenn die enthaltenen Datensätze groß sind. Mit Zeigern kann man auch Bäume und andere beliebige Graphen flexibel implementieren. Bei B-Reps, Octrees und CSG-Bäumen werden wir darauf zurückkommen.

Scanlinien-Flächenfüllen

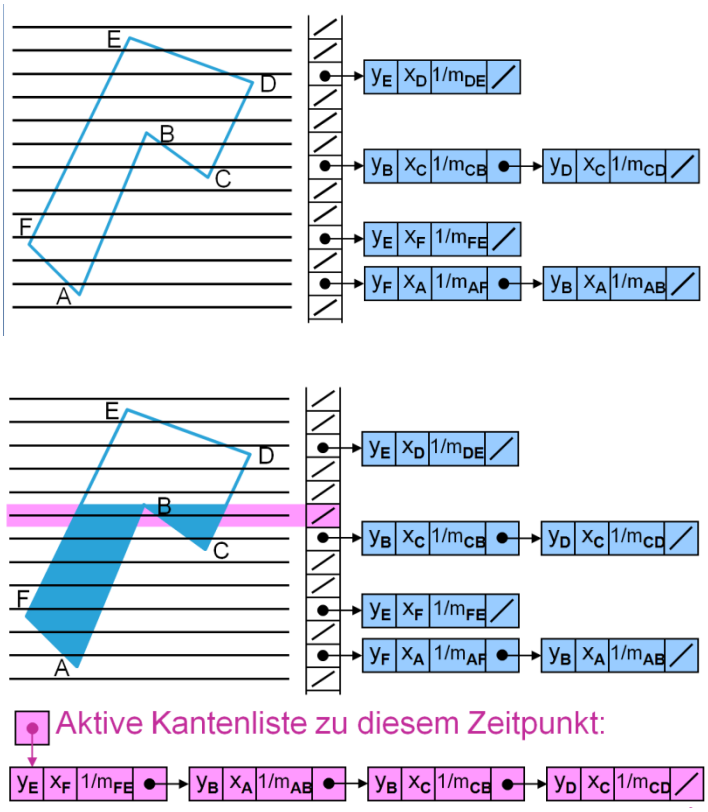
Bei zeilenweisen Füllen eines beliebigen Polygons wird für jede Rasterzeile (Scanlinie) getrennt berechnet, welche Pixel innerhalb und welche außerhalb des Polygons liegen. Bei Anwendung der Odd-Even-Regel wechselt die Eigenschaft „innerhalb“ bei jedem Schnittpunkt einer Polygonkante mit der Scanlinie. Dadurch wird das Füllen innerhalb jeder Zeile ganz trivial: es wird einfach die Scanlinie vom 1. zum 2. Schnittpunkt gefüllt, vom 3. zum 4., vom 5. zum 6. usw.

Inkrementelle Schnittpunktberechnung. Um die Schnittpunkte eines Polygons mit einer Scanline zu berechnen, kann man inkrementell vorgehen. Solche Algorithmen lassen sich auch bei vielen anderen (graphischen und nicht-graphischen) Anwendungen einsetzen, um den Aufwand signifikant zu senken. Grundsätzlich berechnet man dabei die Lösung für eine Bildschirmzeile aus der Lösung der vorhergehenden Zeile, indem so viel wie möglich Information wiederverwendet wird. Dazu kann man sich, wie auch hier beim Flächenfüllen, eine eigene Datenstruktur anlegen, die die benötigten Berechnungen unterstützt.

Wir wollen die Zeilen von unten nach oben bearbeiten. Zuerst legen wir eine geordnete Liste aller Polygonkanten an, wobei als Sortierkriterium der niedrigere der beiden y-Werte der Endpunkte dient. Man speichert bei jeder Kante:

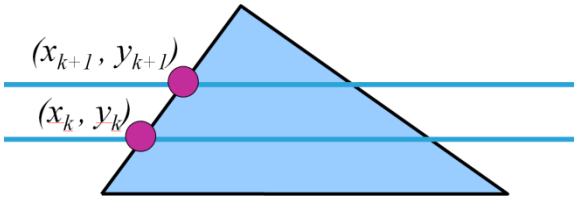
[maximaler y-Wert, x-Wert der 1.Schnittpunktes, inverse Steigung]

Wenn man nun von unten nach oben die Zeilen nacheinander betrachtet, so stehen in dieser sortierten Kantenliste immer die Kanten zuvorderst, die als nächstes für die Schnittpunktberechnung gebraucht werden. Während man die Zeilen von unten nach oben durchläuft hält man eine zweite, kürzere Liste aktuell, die jeweils nur die Kanten enthält, die sich mit der Zeile schneiden ("aktive Kantenliste").



Diese *aktive Kantenliste* erzeugt man inkrementell. Am Beginn sind keine Kanten aktiv. Bei jedem Wechsel von einer Zeile zur nächsten (y → y+1) schaut man, ob die vorderste Kante der sortierten Kantenliste unter diesem neuen y-Wert beginnt. Wenn nicht, dann können auch alle anderen Kanten nicht relevant sein, da deren niedrigster y-Wert ja gemäß Sortierung größer ist als der der ersten Kante. Wenn man aber solche Kanten vorfindet, dann werden die entsprechenden Kanten-knoten in die aktive Kantenliste übernommen, also auch aus der Gesamtliste entfernt. Damit stehen dort dann wieder die Kanten mit den nächstniedrigsten y-Werten. Weiters werden in der aktiven Kantenliste alle Kanten, deren maximales y beim Zeilenwechsel überschritten wurde, entfernt. Dadurch behält man immer eine kurze Liste mit genau den Kanten, die die jeweilige Scanlinie schneiden.

Die aktive Kantenliste wird außerdem immer nach ihren Schnittpunkten von links nach rechts sortiert gehalten, sodass das Zeichnen unmittelbar erfolgen kann. Wie oben erwähnt, braucht jetzt nur die Scanlinie vom 1. zum 2. Schnittpunkt gefüllt werden, vom 3. zum 4., vom 5. zum 6. usw.

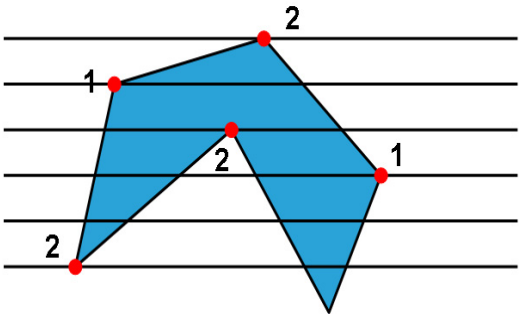


Die Schnittpunkte kann man ebenfalls inkrementell berechnen: Ausgehend vom (exakten) Schnittpunkt (x_k, y_k) einer Scanlinie k mit einer Kante erhält man den nächsthöheren Schnittpunkt (x_{k+1}, y_{k+1}) durch

$$x_{k+1} = x_k + 1/m \quad \text{und} \quad y_{k+1} = y_k + 1.$$

Man sieht nun, wozu der Anstieg 1/m in den Knoten mitgespeichert wird!

Wenn ein Polygoneckpunkt genau auf einer Scanlinie zu liegen kommt, dann muss darauf geachtet werden, dass die Anzahl der Schnittpunkte an dieser Stelle korrekt ist. Manche Punkte müssen dann einfach gezählt werden, andere doppelt (siehe Abbildung). Um diesem Problem aus dem Weg zu gehen werden häufig die Punktkoordinaten um einen kleinen Wert ε nach oben oder unten verschoben. Dieser Wert ist so klein, dass man den Fehler nicht sieht, aber groß genug, dass der Punkt neben der Scanlinie liegt.

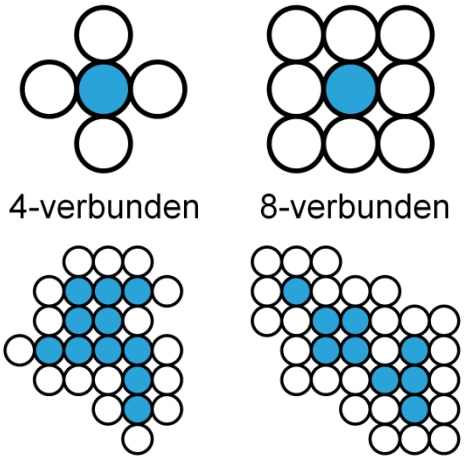


Floodfill-Algorithmus

Ausgehend von einem Startpunkt (Referenzpunkt, Saatpunkt) wird in alle Richtungen gefüllt, bis man an eine Grenze stößt. Diese Grenze kann entweder explizit definiert sein, zum Beispiel durch eine Umrandung in einer bestimmten Farbe, oder aber implizit, z.B. dadurch, dass eine bereits gefärbte Fläche umgefärbt wird. Mischvarianten kommen auch vor. Diese Unterscheidung verändert aber lediglich das Abbruchkriterium beim Füllen. Wir wollen o.B.d.A. (also „ohne Beschränkung der Allgemeinheit“) davon ausgehen, dass die zu füllende Fläche in einer „alten“ Farbe bereits gezeichnet ist und umgefärbt werden soll. Alle anderen Fälle können leicht daraus abgeleitet werden.

4-verbunden versus 8-verbunden

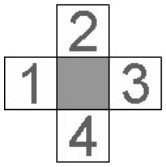
Grundlage für das Weitergehen in alle Richtungen ist die Definition der erlaubten Richtungen. *4-verbunden* heißt, dass eine gültige Verbindung nur über die 4 Hauptrichtungen definiert ist, *8-verbunden* sieht auch diagonale Pixel als verbunden an. 8-verbunden ist also die schwächere Bedingung. Man kann sich leicht überlegen, dass für eine 4-verbundene Fläche eine 8-verbundene Grenze reicht (damit sie nicht „ausrinnen“ kann), eine 8-verbundene Fläche aber eine 4-verbundene Grenzlinie benötigt.

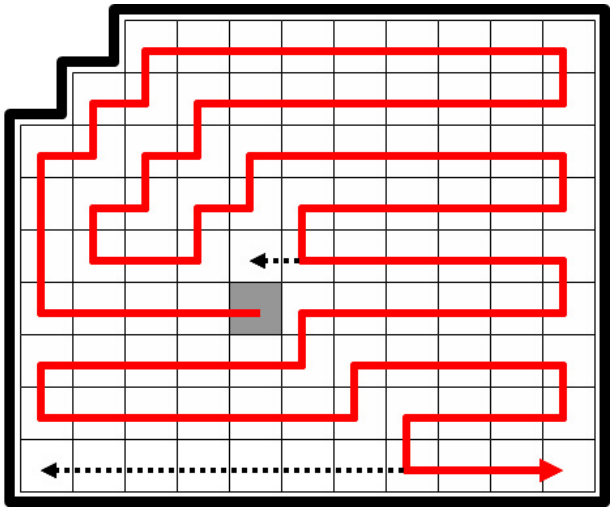


Rekursive Implementierung

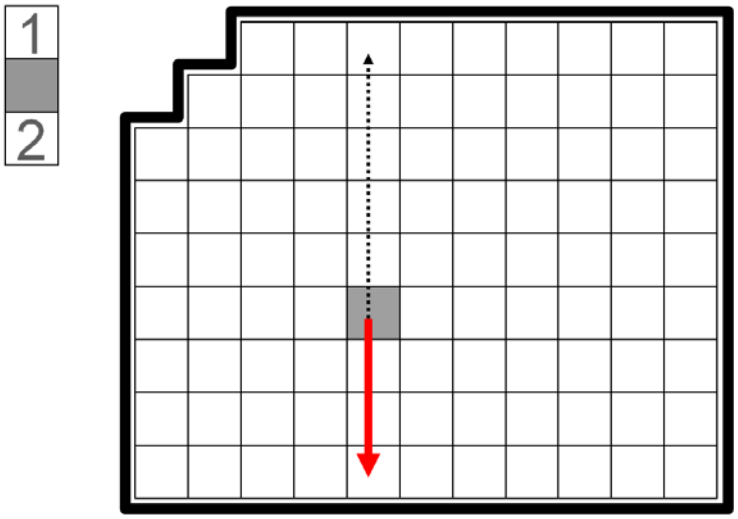
Floodfill für 4-verbundene Flächen lässt sich ganz leicht rekursiv implementieren:

```
void floodFill4 (int x, int y, int new, int old)
{ int color;
    /* set current color to new */
    getPixel (x, y, color);
    if (color = old) {
        setPixel (x, y);
        floodFill4 (x-1, y, new, old);  /* left */
        floodFill4 (x, y+1, new, old);  /* up */
        floodFill4 (x+1, y, new, old);  /* right */
        floodFill4 (x, y-1, new, old)   /* down */
}}
```

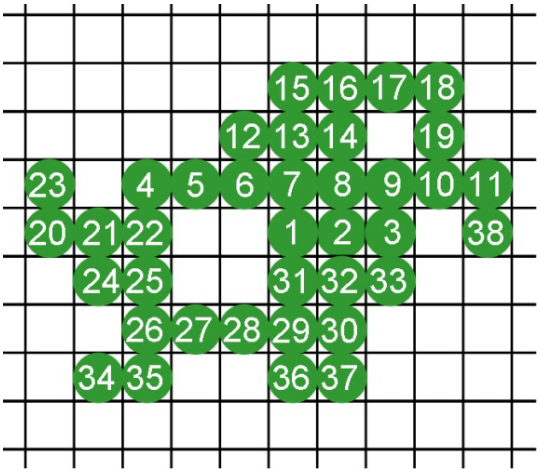




Diese Prozedur erzeugt jedoch eine Füllreihenfolge, die zu einer sehr hohen Rekursionstiefe führt, im allgemeinen Fall bis zur Anzahl der zu füllenden Pixel. In der Abbildung ist links oben die Rekursionsreihenfolge angegeben (soll heißen: zuerst erfolgt der Aufruf nach links, dann nach oben, dann nach rechts, und zum Schluss nach unten), und der durchgezogene Pfeil im Polygon zeigt, wie tief die Rekursion in diesem Fall geht.



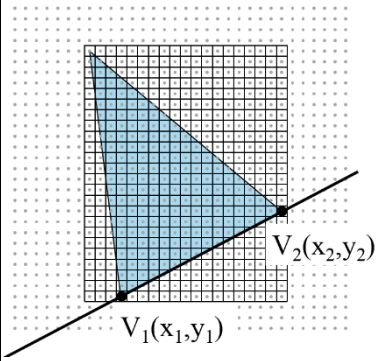
Um das zu verhindern füllt man meist in horizontaler Richtung iterativ und wendet die Rekursion nur nach oben und unten an. Natürlich muss man dabei aufpassen, dass *alle* Spans oberhalb und unterhalb rekursiv aufgerufen werden (ein Span ist eine horizontale nicht unterbrochene Folge von Pixeln, die gemeinsam behandelt werden).



Beispiel

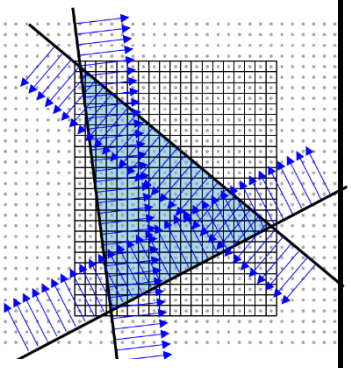
Das nebenstehende Beispiel demonstriert die Füllreihenfolge, wobei eines der Pixel 1, 2 oder 3 als Startpixel gewählt wurde. Die Rekursion geht zuerst nach oben und dann nach unten. Bei jedem Aufruf müssen alle Spans in diese Richtung erwischt werden. Nach dem Span 4-11 wird nach oben von links nach rechts aufgerufen und danach nach unten ebenfalls von links nach rechts. Teile, die bereits gefüllt wurden, werden erkannt, und die Rekursion bricht dort ab (z.B. von 4-11 nach unten ist 1-3 schon fertig). Die Gesamtrekursionstiefe kann zwar theoretisch auch bei diesem Verfahren sehr hoch sein, in der Praxis ist sie aber proportional zur Anzahl der Pixelzeilen (Scanlinien) des Polygons.

Paralleles Füllen konvexer Polygone



Wenn man ausreichend parallele Verarbeitungskapazität zur Verfügung hat (etwa auf einer GPU), kann man konvexe Polygone - also normalerweise Dreiecke – schneller füllen, indem man alle Pixel gleichzeitig bearbeitet. Es wird für jedes Pixel unabhängig von allen anderen entschieden, ob es gefüllt werden muss. Der zu untersuchende Bereich wird dabei auf das engste Rechteck eingeschränkt, das das Polygon umschließt. Aus den Eckpunkten $V_1(x_1,y_1)$ und $V_2(x_2,y_2)$ einer Polygonkante berechnet man deren Trägergerade:

$$(y_1-y_2)x + (x_2-x_1)y + x_1y_2-x_2y_1 = 0.$$



Pixel (x,y), deren Mittelpunkt in die Gleichungen aller Kanten eingesetzt einen Wert >0 ergeben, liegen innerhalb des Polygons, alle anderen liegen außerhalb.

Behandlung von 3D Polygonen

3D Polygone sind normalerweise Oberflächenelemente von Objekten im Raum. Dementsprechend sind deren Attribute auch hauptsächlich Eigenschaften der Oberfläche dieses Objektes: Farbe, Materialparameter (Rauheit, Absorption, …), Transparenz, Textur, geometrische Mikrostruktur, Reflexionsverhalten usw., die in Zusammenspiel mit einer definierten Beleuchtung szenenabhängige Effekte liefern. Zusätzlich werden an den Eckpunkten oft Normalvektoren und Texturkoordinaten angegeben, um eine schnelle und adäquate Berechnung des Aussehens des Polygons zu ermöglichen. Wir werden später sehen, wie man das alles effizient einsetzt.